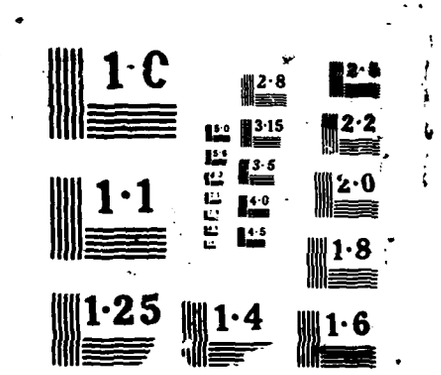
AD-A188 225 THE BEHAVIOR OF SHARED OBJECTS: CONCEPTS PITFALLS AND A 1/1
NEW MODEL(U) NORTH CAROLINA UNIV AT CHAPEL HILL DEPT OF
COMPUTER SCIENCE J STAUNSTRUP ET AL 15 OCT 87
UNCLASSIFIED N00014-86-K-0680 F/G 12/5 NL


1·C
1·1
1·25
2·8
3·15
3·5
4·0
4·5
1·4
2·2
2·0
1·8
1·6

Contract N00014-86-K-0680

The behavior of shared objects: concepts, pitfalls, and a new model.

Jørgen Staunstrup
Datalogisk Afdeling
Aarhus Universitet
DK-8000 Aarhus C
Danmark

Jurg Nievergelt
Department of Computer Science
University of North Carolina
Chapel Hill, N.C. 27514
USA

October 15, 1987

Keywords: specification, concurrency, shared registers, arbiters, distributed systems.

Abstract:

We discuss the behavior of objects shared by several concurrent processes whose operations on the objects may overlap in time. Shared objects range from files in a distributed database to a piece of hardware, e.g. a wire or a flip-flop. This wide range of operations share some common properties and fundamental limitations. These are of primary importance in understanding the behavior of shared objects.

The specification of the behavior of shared objects under concurrent access includes both functional and timing aspects. It is tempting to ignore or oversimplify timing, for example by assuming that all operations have bounded response times, but this may lead to contradictions. We propose a new specification technique, based on **predictor automata**, which captures both functional and timing behavior.

1 Motivation

We discuss the behavior of objects shared by several concurrent processes whose operations on the objects may overlap in time. A shared object may take many different forms, e.g. a file in a distributed database, a peripheral device, or a communication line.

We consider two simple objects, a register and an arbiter, which can be found on all levels of a distributed system, from the hardware to applications software.

A shared register supports two operations: `write(v)` (store the value v in the register), and `read` (return the value last stored). A register may be a record of a file, it may be a storage cell, or it may be a single bit. A correct sequential execution of these operations satisfies the axiom:

$$\{TRUE\}\ \texttt{write(v)}; \texttt{r} := \texttt{read}; \{v = r\}$$

When different operations on a register may overlap in time, the desired semantics is less clear. What value should a `read` return when it overlaps with a `write`? The simple solution of queuing all operations in order of arrival has several undesirable consequences. In a multiprocessor, and particularly in a geographically distributed system, the order of arrival may not be exactly determinable. Even if it is, a central queue may create a severe bottleneck. It is, therefore, natural to consider a register that allows operations which overlap in time. This raises the question of what a "correct behavior" of such a register is, as the following options show: 1) delay a read until all writes are completed, 2) return a value distinct from all other values, to indicate that the register is in a transient state, 3) return an arbitrary value. The last option leaves an implementor much more freedom than the two first. He may face a variety of environments:

- a system without a common notion of time, where it may be impossible to decide which of two operations came first,
- a system without control of the underlying hardware or software, which may rearrange the operations,
- a system which cannot guarantee a bound on the time needed to complete an operation,
- a system which optimizes the speed of some operations at the expense of others; therefore, their duration may be orders of magnitude different.

The value returned by a read may be arbitrary at times, but must satisfy the semantics of sequential execution when no operations overlap. In order to disqualify trivial implementations (e.g., always return the same value regardless of what has been written into the register) we need to specify exactly when we do and do not care about the value returned. The next section describes several previously published specifications.

Two kinds of behavior must be distinguished: **functional** and **timing** behavior. The first deals with "what" values are returned, the latter with "when" they are returned. Several recent publications [4] [6] emphasize the functional behavior. In the next section we analyze these definitions and show that contradictions may result from ignoring the temporal behavior.

The above discussion was cast in the terminology of a "distributed system" existing on top of several layers of software and hardware. This is the scenario most programmers face when dealing with shared objects. Exactly the same issues must be addressed at the lowest levels of hardware design. Here, the problems emerge when different parts of the hardware coordinate their use of shared circuitry, which is necessary for any kind of communication (at the very least a wire must be shared). In the next few sections we use terminology aimed at this low level of hardware. This is to stress the fundamental nature of the problems, and to avoid the unrealistic assumption, that lower levels can be called upon to solve the problem.

2 Behavior of shared registers

The following three kinds of shared registers, have been defined by Lamport [4].

- **A safe register:** a read not overlapping any write returns the value last written, and a read always returns a valid value, i.e. a value in the domain of the register (0 or 1 in the case of 1-bit registers).
- **A regular register:** a safe register where a read overlapping a write must return either the new or the old value of the register.
- **An atomic register:** reads and writes behave as if they were executed in some serial order.

All three types of registers assumed that no process has to wait. The safe register is the weakest and therefore also the simplest to implement. A regular register is stronger than a safe one; but not as strong as the atomic register. Further characterizations of a register can be made by defining its width and the number of readers and writers it may have. Lamport argues that a 1-writer/1-reader/1-bit safe register is the weakest that serves as a useful notion of computation.

Recently a number of papers have shown how registers that meet strong requirements can be simulated with weaker ones [1] [2]. For example, an n-reader/1-writer atomic register may be built from 1-reader/1-writer safe registers (using many safe shared registers

to simulate one atomic shared register). All these papers claim that the algorithms work *without waiting* (they use no busy waiting or other synchronization mechanisms). An algorithm is said to be without waiting, if it uses a constant number of read or write operations (on the weaker register used in the simulation). This is called the **bounded response time assumption.**

[1] shows that a 2-writer/n-reader atomic register may be built out of 1-writer/1-reader atomic registers. Hence, safe registers are powerful enough to build any other type of register. This "final solution" points to the safe register as the basic building block of a distributed system. We see a fundamental contradiction, however, in assuming the existence of a 1-bit safe register with bounded response time.

3 The behavior of arbiters

A fundamental limitation of any physically shared object has been known for some time [3][9]. To illustrate it, consider an **arbiter** which is a shared object with four operations *request*1, *request*2, *grant*1, and *grant*2. The arbiter grants permission to at most one among two processes that request it. In case of conflict, it does not matter which request is granted (fairness is not an issue for this discussion). The arbiter must, however, be **responsive**: if only one process requests the resource, it gets the grant. There seems to be common agreement that an arbiter which is guaranteed to make a decision in bounded time *cannot* be built (out of electronic components). No matter how long one waits, there is a non-zero probability that the arbiter has not yet made up its mind (for example, when requests from two processes arrive so closely in time that they appear simultaneous). Marino [5] has proved this under the assumption that all components can be described by smooth functions, and [3] provides experimental verification. An arbiter that may, with very low probability, wait arbitrarily long before making a decision is still a practically useful device: a linear increase in waiting time gives an exponential decrease in the probability of finding the arbiter still undecided.

4 A contradiction

A comparison of the behavior of the shared objects discussed, the register and the arbiter, leads to paradox.

Several algorithms exist for constructing an arbiter out of 2-writer/2-reader atomic shared registers. The first, Dekker's algorithm, has since been simplified and extended

in a number of papers, e.g. [8]. The key idea common to all algorithms is that an atomic register can be used to break a tie between two requests. For example, by letting a request operation write the identity of the process into an atomic register (often called *turn* in the published algorithms).

Recall that all the shared register constructions referenced above involved no waiting. It follows that an arbiter which is guaranteed to make a decision in bou nded time can be built from safe 1-bit shared registers. This *contradicts* the argument of section 3 that such an arbiter cannot be built. Where is the error? We believe the assumption of the existence of 1-bit safe shared registers with bounded response time is at fault. In other words, asynchronous access to physically shared registers implies:

- either unbounded waiting for valid values,
- or bounded waiting for values that may be invalid.

An invalid bit looks high to some gates and low to others. This awkward physical reality is avoided by waiting for the value to settle, which may delay an arbiter indefinitely. Because the probability of an operation taking a very long time can be made negligible, practical hardware design typically ignores this effect. As the design of high speed asynchronous circuitry pushes the boundary of the feasible, a good theoretical specification of shared objects with unbounded response times will be useful. We propose such a model in the next section.

5 Operations with unbounded response time

We have argued that the assumption of bounded response time is unrealistic, that in any consistent implementation of concurrent operations on shared data, the act of arbitrating exclusive access, of "tipping the scales", can take an indefinite length of time. In a good implementation, the probability of a long delay will decrease rapidly with increasing response time, but need never become zero.

Several techniques have been developed to define the semantics of concurrent operations under the assumption of bounded response time [4][6]. The case of unbounded response time, on the other hand, does not appear to have been studied. We present an approach to this problem, called **probabilistic state transition analysis** based on **predictor automata**, a generalization of finite state machines. Predictor automata model the behavior of an object under concurrent access at a similar level of generality as finite automata model sequential circuits. The model is mathematically simple: it uses

concepts and techniques from elementary probability theory, linear algebra, and finite automata. No physical assumptions are built into the model; rather, it is sufficiently general to model objects with a wide range of physical behaviors. As an example, we show how the same logical object can be modeled to exhibit different temporal behavior during the transients between stable states, where each of these behaviors reflects a different physical implementation of the object. Probabilistic state transition analysis is a further development of a technique developed in [7] to define the semantics of bounded response time concurrency.

A predictor automaton P is given by:

- a finite set $S = \{s_1, ..., s_n\}$ of **states**
- a finite set $O = \{o_1, ..., o_m\}$ of **operations** (inputs)
- m **transition matrices**: $M^1, ..., M^m$, where $M^k = [f_{ij}^k(t)]$, and for each triple (k, i, j) with $1 \leq k \leq$ m and $1 \leq i, j \leq n$: $f_{ij}^k(t)$ is a real-valued function of, $t \geq 0$ that satisfies all the following conditions:
  1. $f_{ij}^k(0) = 1$ for i = j and $f_{ij}^k(0) = 0$ for $i \neq j$
  2. $0 \leq f_{ij}^k(t) \leq 1$
  3. $\forall k, i : \sum_{j=1}^{j=n} f_{ij}^k(t) = 1$.

The definition is motivated by the following interpretation. Consider an object which at any moment is in one of a finite set of states: $s_1, ..., s_n$. At certain quiescent times we may know the precise state of the object. When an input (an operation) is applied, the object "gets ready" to change state according to a transition function $f : S \times O \rightarrow S$. In contrast to conventional finite automata, we do not assume that this transition occurs instantaneously; instead we assume that the transition may occur at any time after the input is applied.

During the transients between stable states our knowledge of the object's current state will be incomplete; but we aim to know the probability of it being in any one state. Thus we introduce the concept of a **probabilistic state** as a vector, $(p_1, ..., p_n)$, where $p_i \geq 0$ is the probability of the object being in state s_i, and $\sum p_i = 1$.

We interpret $f_{ij}^k(t)$ as the probability of finding the object in state s_j at time t, given that it was in state s_i at time 0 when input o_k was applied, and that no other input has been applied since $t = 0$. The intuitive significance of the conditions above can now be stated as: 1) no instantaneous change; 2), 3) at any time t, the object is in some state

In typical cases we might assume that the intended transition will indeed occur at some time t by chosing functions $f_{ij}^{k}(t)$ that tend to 1 as t tends to infinity, perhaps even monotonically. But we do not require this as a part of the definition, in order to be able to model erratic behavior which, although undesirable, might occur and needs to be studied.

5.1 Example: flip-flop

Consider a flip-flop with the two states *high* and *low*, and a probabilistic state $(p_{high}, p_{low}) = [p, 1-p]$. Consider the operations *set* (to *high*), *reset* (to *low*), and *negate* (complement). Assume an object with the following physical behavior.

Persistence: If the operation does not require a change of state, the object remains in its present state (e.g. there is no read/write cycle that temporarily destroys the state before restoring it to the old value). This implies for the operations *set* and *reset*: $set_{high\ high}(t) = 1$, and $reset_{low\ low}(t) = 1$.

Condition 3), which requires that each row of a transition matrix adds up to 1, provides an equivalent way of stating the same requirement:

No spontaneous transitions. The object will not switch to any new state unless an operation requests such a state change. This implies for *set* and *reset*: $set_{high\ low}(t) = 0$, and $reset_{low\ high}(t) = 0$.

Uniform monotonic transitions: If the operation requires a change of state from any state s_i to any state $s_j \neq s_i$, the probability of finding the object in the target state s_j increases monotonically from 0 to 1 according to a switching function, $f(t)$, which is independent of the source state s_i, the target state s_j, and the operation o_k. This implies:
$set_{low\ high}(t) = reset_{high\ low}(t) = negate_{low\ high}(t) = negate_{high\ low}(t) = f(t)$;
$set_{low\ low}(t) = reset_{high\ high}(t) = negate_{low\ low}(t) = negate_{high\ high}(t) = 1 - f(t)$.

The following vector*matrix multiplications show how each of the operations transforms the probabilistic state of this flip-flop.

set :

$$[p, 1-p] * \begin{vmatrix} 1 & 0 \\ f(t) & 1-f(t) \end{vmatrix} = [p + (1-p)f(t), (1-p)(1-f(t))]$$

reset :

$$[p, 1-p] * \begin{vmatrix} 1-f(t) & f(t) \\ 0 & 1 \end{vmatrix} = [p(1-f(t)), pf(t)+1-p]$$

negate :

$$[p, 1-p] * \begin{vmatrix} 1-f(t) & f(t) \\ f(t) & 1-f(t) \end{vmatrix} = [p+(1-2p)f(t), (1-p)-(1-2p)f(t)]$$

Notice that with $f(\infty) = 1$ these equations describe the logically correct behavior of a flip-flop in its stable states.

5.2 Extensions and comparisons

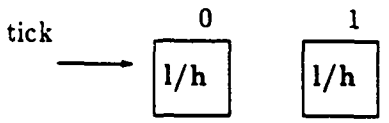
It is of course possible to define standard concepts such as initial state and final states (a deterministic state s_i being the special case $p_i = 1$ of a probabilistic state $(p_1, ..., p_n)$), output function, and a language (a set of input strings) being accepted. We omit these concepts as unnecessary for the present purpose.

Predictor automata differ from finite automata primarily by using the concept of probabilistic state, and by replacing a deterministic transition function $f : S \times O \rightarrow S$ by probabilistic state transitions. Both of these generalizations have been made in a number of studies on probabilistic automata. Predictor automata differ from the latter by incorporating the notion of real time, as opposed to an artificial concept of time measured by counting how many inputs have occurred. The name "predictor automata" is chosen to express that: in the absence of any further inputs (disturbances) the object may still undergo unknown transitions, but we have a probabilistic prediction of its state at any future time.

The concept of real time allows us to model concurrent operations as those whose duration overlaps in time. In other words, a new input may be applied before the transients caused by the preceding input have died out. Introducing real time does not imply that different processes that share an object can read or otherwise use this time; in particular, we assume no common clock. Real time is merely a tool for analyzing the behavior of an object shared by concurrent processes.

5.3 Example: counter

A 0-1-2-3 counter consisting of two bits may be thought of as the series composition of two one bit counters. A single operation *tick* increments the counter.



This object has four states: ll, lh, hl, and hh, so a probabilistic state is $(p_{ll}, p_{lh}, p_{hl}, p_{hh})$ (where $p_{ll} + p_{lh} + p_{hl} + p_{hh} = 1$). The operation *tick* is described by the matrix:

$$tick: \begin{vmatrix} 1-f(t) & f(t) & & \\ & 1-f(t) & f(t) & \\ & & 1-f(t) & f(t) \\ & & & 1 \end{vmatrix}$$

This counter gets stuck in the state hh once it gets there. A more useful modulo 4 counter is obtained by replacing the last row of the matrix by the vector: $|f(t),\ 0,\ 0,\ 1-f(t)|$ The transition matrices of both counters are compositions of corresponding matrices for a flip-flop. In addition to series composition, parallel composition of predictor automata can be defined in a way similar to well-known composition rules for finite state automata.

6 Behavior of shared objects

We propose the flip-flop specified above as an alternative to the safe register. We may use a convention such as $p_{high} > threshold$ to model the flip-flop's state high, and similarly for the state low. The response time of the object may be defined as the smallest t such that $p_{high}(t) > threshold \vee p_{low}(t) > threshold$.

The arbiter can be modeled by a flip-flop with three states: *low*, *undecided*, and *high*. A request from process 1 attempts to set the state to *low*, a request from process 2 attempts to set the state to *high* and a release will set the state to *undecided*. The probabilistic state is: (p_l, p_u, p_h). The operations may be specified as follows:

$$request1: \begin{vmatrix} 1 & & \\ f(t) & 1-f(t) & \\ & & 1 \end{vmatrix} \quad release1: \begin{vmatrix} 1-f(t) & f(t) & \\ & 1 & \\ & & 1 \end{vmatrix}$$

$$request2: \begin{vmatrix} 1 & & \\ & 1-f(t) & f(t) \\ & & 1 \end{vmatrix} \quad release2: \begin{vmatrix} 1 & & \\ & 1 & \\ & f(t) & 1-f(t) \end{vmatrix}$$

For simplicity's sake, the same function f is used to describe the transition from the undecided state to high (low) and the transition back to undecided. [9] argues that $f(t) = 1 - e^{-ct}$ is a realistic approximation for an electronic circuit.

Note that the above does not describe an arbitration algorithm - it is merely a specification of the behavior of a physically shared object used in such an algorithm.

7 Conclusion

The first part of this paper pointed out that *unbounded response time is an inherent property of all operations on shared objects.* Concurrent processes experience delays not only because one lucky process with access to a shared resource locks out others, but also because the act of tipping the scales among processes competing for access takes time. Section 4 shows that delays are inherent in any computation involving physically shared objects.

The second part proposes a model, predictor automata, for describing the behavior of shared objects under concurrent operations with unbounded response time. The model is simple and technology-independent because it avoids introducing intermediate physical states. Instead, it models transient behavior by the probability of being in any one logical state.

Acknowledgement

We thank Mark Greenstreet for many educating discussions on concurrency. The first author was supported in part by the National Science Foundation under Grant No. CCR-8619663 while visiting the Department of Computer Science, University of Washington, Seattle. This work was also supported by ONR under contract N00014-86-K-0680.

References

[1] B. Bloom, Constructing Two-Writer Atomic Registers, **Proceedings from PODC**, pp. 249-259, ACM, Vancouver August 1987.

[2] J.E. Burns and G.L. Peterson, Constructing Multi-reader Atomic Values from Non-atomic values, **Proceedings from PODC**, pp. 222-231, ACM, Vancouver 1987.

[3] T.J. Chaney and C.E. Molnar, Anomalous behavior of synchronizer and arbiter circuits. **IEEE Transactions on Computers, vol. 22**, April 1973, pp. 421-422.

[4] L. Lamport, On interprocess communication, **Distributed Computing**, vol. 1, pp. 86-101, Springer Verlag 1986.

[5] L.R. Marino, General Theory of Metastable Operation, **IEEE Transaction on Computers, vol. 30**, no. 2, pp. 107-115, February 1981.

[6] J. Misra, Axioms for Memory Access in Asynchronous Hardware Systems, **ACM Tr. on Programming Languages and Systems, 8**, pp. 142-153, 1986.

[7] J. Nievergelt and J. Staunstrup, What is a Correct Behavior of a File under Concurrent Access ?, **Proceedings Second International Symposium on Distributed Databases**, H. J. Schneider (ed.), 93-103, North Holland 1982.

[8] G.L. Peterson, Myths about the mutual exclusion problem, **Information Processing Letters 12**, pp. 115-116, 1981.

[9] C.L. Seitz, System Timing, chapter 7 in **Introduction to VLSI systems** by C. Mead and L. Conway, Addison-Wesley 1978.

END

DATE

FILMED

3-88

DTIC